Signals

Сигналы являются способом передачи уведомлений между процессами или между ядром системы и процессами.
Уведомление о некотором произошедшем событии.
Сигналы можно рассматривать, как простейшую форму межпроцессного взаимодействия, хотя на самом деле они больше напоминают программные прерывания, которые вторгаются в ход нормального выполнения процесса.

Событие, порождающее сигнал, может быть действием пользователя, может быть вызвано другим процессом или ядром.

Сигналы используются, как простейшее, по мощное средство межпроцессного взаимодействия.

Действия, вызываемые для обработки сигналов, являются принципиально *асинхронными*.

Каждый сигнал имеет уникальное символьное имя и соответствующий ему номер.
Например, сигнал прерывания, посылаемый процессу при нажатии пользователем клавиш <Ctrl>+<C>, имеет имя SIGINT.
Сигнал, генерируемый комбинацией <Ctrl>+<\>, называется SIGQUIT.
Современные версии Linux насчитывают несколько десятков различных сигналов.

Сигнал может быть отправлен процессу либо ядром, либо другим процессом с помощью системного вызова *kill()*.
Первым параметром вызова *kill()* является PID процесса, которому данный сигнал предназначен,
а вторым параметром указывается какой именно сигнал надо отправить.

В число ситуаций при которых ядро отправляет процессу (или целой группе) определенные сигналы, входят и так называемые аппаратные *особые ситуации*.
Например деление на ноль, обращение к недопустимой области памяти и др.
В зависимости от типа сигнала (события его породившего), реакцией системы на его получение (обработкой по умолчанию) может быть:
Exit – выполнение действий совпадающих с семантикой системного вызова *exit()*;
Core – создание файла снимка текущего состояния ядра, затем выполнение *exit()*; файл *core,* хранящий образ памяти процесса, может быть впоследствии проанализирован программой отладчиком для определения состояния процесса непосредственно перед завершением;
Stop – остановка и подвешивание процесса;
Ignore – игнорировать, отбросить сигнал.

Вместо того, чтобы предоставлять системе самой обрабатывать

сигнал, процесс может обладать собственным *обработчиком*, выполняющим при получении сигнала какие-либо специфические действия
в контексте требований самого этого процесса.
Если для сигнала устанавливается функция-обработчик, то говорят, что сигнал перехватывается (относительно стандартного действия).

Для организации собственной обаботки используется вызов *signal() .*
Системный вызов *signal() инициализирует* собственный обработчик сигнала,
номер которого указан первым параметром вызова.
Второй параметр вызова *signal()* указывает на этот собственный обработчик.

Программа *signal_catch* демонстрирует функционирование собственных обработчиков сигналов SIGINT (отправляем нажатием Ctrl-C) и
SIGQUIT (отправляем нажатием Ctrl-\).

```
/* Программа signal_catch.cpp */
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<signal.h>
main(void)
{
int     i;
void   signal_catcher(int);
if(signal(SIGINT, signal_catcher)==SIG_ERR){
    perror("SIGINT");
    exit(1);
    }
if(signal(SIGQUIT, signal_catcher)==SIG_ERR){
    perror("SIGQUIT");
    exit(2);
    }
for(i=0; ;++i){             /* Зацикливание */
    printf("%i\n", i);        /* Индикация счетчика */
    sleep(1);
    }
}
void
signal_catcher(int the_sig){
    signal(the_sig, signal_catcher);    /* Сброс */
    printf("\nSignal %d received.\n", the_sig);
    if(the_sig==SIGQIUT)
    exit(3);
}
```

Рассмотрим некоторые виды сигналов
и реакции системы на их возникновение:

Название	***Действие по умолчанию***	***Значение сигнала***
SIGILL	Завершить +core	Сигнал посылается ядром, если процесс попытался выполнить недопустимую инструкцию.
SIGINT	Завершить	Сигнал посылается ядром всем процессам текущей группы при нажатии клавиши прерывания (<Ctrl> или <Ctrl>+<C>)
SIGKILL	Завершить	Сигнал, при получении которого выполнение процесса завершается. Этот сигнал нельзя ни перехватить, ни игнорировать.
SIGPIPE	Завершить	Сигнал посылается при попытке записи в канал или сокет, получатель данных которого завершил выполнение (закрыл соответствую- щий дескриптор).
SIGPOLL	Завершить	Сигнал отправляется при наступлении определенного события для устройства, которое является опрашиваемым.
SIGPWR	Игнорировать	Сигнал генерируется при угрозе потери питания. Обычно отправляется, когда питание системы переключается на ИБП (UPS).
SIGQUIT	Завершить +соге	Сигнал посылается ядром всем процессам текущей группы при нажатии <Ctrl>+<\>.
SIGSTOP	Остановить	Сигнал отправляется всем процессам текущей группы при нажатии <Ctrl>+<Z>. Получение сигнала вызывает останов выполн процесса.
SIGTERM	Завершить	Сигнал обычно представляет предупреждение, что процесс вскоре будет уничтожен, что позволяет процессу подготовиться к этому - удалить временные файлы, завершить необходимые транзакции и т. д.
SIGTTIN	Остановить	Сигнал генерируется ядром (драйвером терминала) при попытке процесса фоновой группы осуществить чтение с терминала.
SIGTTOU	Остановить	Сигнал генерируется ядром (драйвером терминала) при попытке процесса фоновой группы осуществить запись на терминал.
SIGUSR1	Завершить	Сигнал предназначен для прикладных задач как простейшее средство межпроцессного взаимодействия.
SIGUSR2	Завершить	Сигнал предназначен для прикладных задач как простейшее средство межпроцессного взаимодействия.

Системный вызов *signal()* организует обработк*у* сигнала отличную от указанной по умолчанию, т.е. определяет новую *диспозицию* сигнала, в качестве которой может быть:
- пользовательская функция-обработчик;
либо одно из следующих значений:

SIG_IGN - указывает ядру, что сигнал следует игнорировать,
SIG_DFL - указывает, что при получении процессом сигнала необходимо вызвать системный обработчик, т.е. выполнить действие по умолчанию.

В случае необходимости измененная (новая) диспозиция сигнала может быть восстановлена. Это достигается благодаря тому, что системный вызов *signal()* при успешном выполнении возвращает ту диспозицию, что была
до изменения.

Signals handling

Программа *signal_alarm* содержит собственный обработчик сигнала SIGALRM от системных часов.

```
/* Программа signal_alarm.cpp */
/* Иллюстрация использования SIGALRM, setjmp и longjmp */
/* для воплощения тайм-аута */
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<setjmp.h>
#include<signal.h>
main(void)
{
char    buffer[100];
int     v;
while(1){
   printf("enter a string:");
   v = t_gets(buffer, 5);
   switch(v){
        case -1:        exit(1);/* Возможно EOF */
        case -2:        printf("timed out!\n");
        break;
        default:        printf("you typed %d caracters\n", v);
        }
   }
}
```

```
jmp_buf  timeout_point;
                    /* Это обработчик сигнала SIGALRM */
void timeout_handler(int sigtype)
{
longjmp(timeout_point, 1);
}
   /* This is the important bit */
int t_gets(char *s, int t)
                    /* Буфер для значения тайм-аута в секундах */
{
char *ret;
signal (SIGALRM, timeout_handler);
if(setjmp(timeout_point) != 0)
   return -2;                       /* Тайм-аут */
alarm(t);
ret = gets(s);
alarm(0);        /* Снять звуковой сигнал */
if (ret==NULL) return -1;      /* EOF */
   else return strlen(s);
}                   /* Завершение  ^D */
```

IPC

Система IPC (*Inter Process Communications*) представлена в ОС UNIX следующими средствами:

- очереди сообщений (*Message Queue*),
- семафоры (*Semaphores*) и
- разделяемая память (Shared Memory).

Объекты IPC используются совместно произвольными процессами и могут оставаться существовать в системе даже после завершения этих процессов. Поэтому процедура назначения имен объектов IPC является более сложной, чем просто указание имени, как это возможно в случае, например, обычного файла.
Имя для объекта IPC называется ключом (*key*) и
генерируется функцией *ftok()* из двух компонентов :
имени файла и идентификатора проекта

```
#include <sys/types.h>
#include <sys/ipc.h>
...
key_t  ftok (char *filename, char proj);
```

В качестве *filename* можно использовать имя некоторого файла, известное всем взаимодействующим процессам.
Например, это может быть имя программы-сервера, к которой обращаются клиенты, и оно им заранее известно. Важно, чтобы этот файл существовал на момент создания ключа. Также нежелательно использовать имя файла, который создается и удаляется в процессе работы приложения,
поскольку при генерации ключа используется номер файла.
Вновь созданный файл может иметь другой *inode* и впоследствии процесс, желающий иметь доступ к объекту, получит неверный ключ.

Пространство имен позволяет создавать и совместно использовать IPC *не родственным процессам*.
Для ссылок на уже созданные объекты используются идентификаторы, точно так же, как файловый дескрипторы используется для работы с файлами, открываемыми по имени.

Каждое из разновидностей IPC средств имеет свой уникальный идентификатор (дескриптор), используемый ядром для работы с объектом. Уникальность дескриптора обеспечивается только для каждого из типов объектов IPC (очереди сообщений, семафоры и разделяемая память),
т.е. какая-либо очередь сообщений может иметь тот же численный идентификатор, что и разделяемая область памяти (хотя любые две очереди сообщений должны иметь различные идентификаторы).

Работа со всеми тремя видами IPC средств в определенной степени *унифицирована*.
Так, для создания или получения доступа к объекту используются

соответствующие системные вызовы *get()*:
msget() - для очереди сообщений,
semget() - для семафора и
shmget() - для разделяемой памяти.
Все эти вызовы возвращают дескриптор объекта в случае успеха и -1 , в случае неудачи.
Отметим, что функции *get()* позволяют процессу только получить ссылку на объект. Конкретные же операции над объектом (помещение или получение сообщения из очереди сообщений, устанавливка семафора или запись данных в разделяемую память) производятся с помощью других системных вызовов, также в унифицированной манере.

Все функции *get()* в качестве аргументов используют ключ *key ,* а также флажки создания объекта *ipcflag*.
Остальные аргументы зависят от конкретного типа IPC объекта.
Переменная *ipcflag* определяет права доступа к объекту, а также указывает, создается ли новый объект (*IPC_CREAT*) или требуется доступ к существующему (*IPC_EXCL*).

Работа с объектами IPC во многом похожа на работу с файлами, однако , одним из различий является то, что дескрипторы обычных файлов имеют значимость в контексте процесса. Так файловый дескриптор 100 одного процесса в общем случае никак не связан с дескриптором 100 другого *неродственного* процесса (т. е. эти дескрипторы ссылаются на различные файлы).
В то же время как значимость дескрипторов объектов IPC распространяется на всю систему. Например, процессы, использующие одну и ту же очередь сообщений, получат одинаковые дескрипторы этого объекта.

Для каждого из созданных объектов IPC ядро поддерживает соответствующую *внутреннюю* (системную) *структуру* данных.

Система не удаляет созданные объекты IPC даже тогда, когда ни один процесс не пользуется ими.
Удаление созданных объектов является обязанностью самих процессов, которым для этого предоставляются соответствующие функции управления :
msqctl(),
semctl(),
shmctl().
С помощью этих функций процесс может получить и установить ряд полей *внутренних структур*, поддерживаемых системой для объектов, а также удалить созданные объекты.
Безусловно, при совместном использовании объектов IPC процессы предварительно должны "договориться", какой именно процесс и когда удалит ставший ненужным объект. обычно таким процессом является процесс-сервер.

Message Queue

Очереди сообщений являются составной частью IPC средств
Процессы могут записывать и считывать сообщения
из различных очередей.
Процесс, пославший сообщение в очередь, может
не ожидать чтения этого сообщения каким-либо другим процессом.
Он может завершить свое выполнение, оставив в очереди сообщение,
которое будет прочитано другим процессом позже.

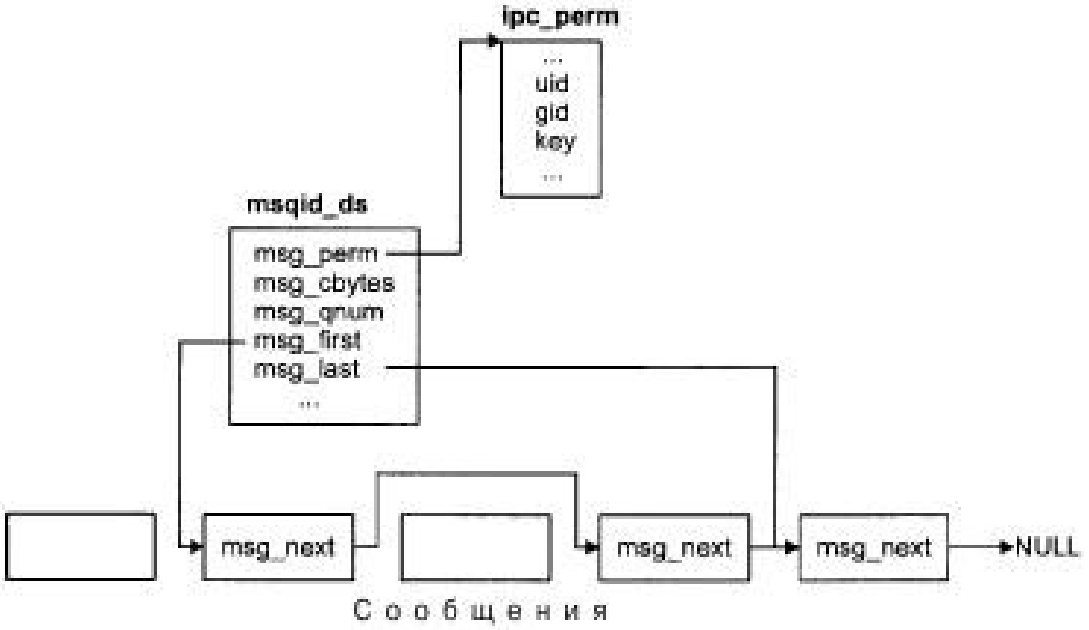
Процессы могут обмениваться структурированными данными,
имеющими следующие атрибуты:
- тип сообщения (позволяет мультиплексировать сообщения в одной очереди);
- длина данных сообщения в байтах (может быть нулевой);
- собственно данные (если длина ненулевая, могут быть структурированными).

В примере *gener_mq* создается пять очередей сообщений, затем вызовом *popen()* выполняется *shell* команда *ipcs* ,
выводящая на консоль список всех имеющихся на данный момент
IPC ресурсов (в том числе *msg*) и их атрибуты.
После этого все очереди сообщений удаляются.

```
/* Программа gener_mq.cpp */

/* Создание очереди сообщений */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX 5

main(void){
FILE *fin;
char  buffer[PIPE_BUF];
char  u_char = 'A';
int     i, n, mid[MAX];
key_t key;
for (i=0; i<MAX ; ++i, ++u_char){
   key = ftok(".", u_char);       /* Генерация ключа и создание ресурса */
        if ((mid[i] = msgget(key, IPC_CREAT | 0660))==-1){
        /* IPC_CREAT – создавать новую, даже если msg уже имеется */
            perror("Queue create");
            exit(1);
        }
   }
```

```
fin = popen("ipcs", "r");/* Запуск ipcs команды */
while((n = read(fileno(fin), buffer, PIPE_BUF))>0)
   write(fileno(stdout), buffer, n);
                          /* Вывод команды ipcs */
   pclose(fin);
for (i=0; i<MAX; ++i)
   msctl(mid[i], IPC_RMID, (struct msgid_ds *)0);
                          /* Удаление */
   exit(0);
}
```

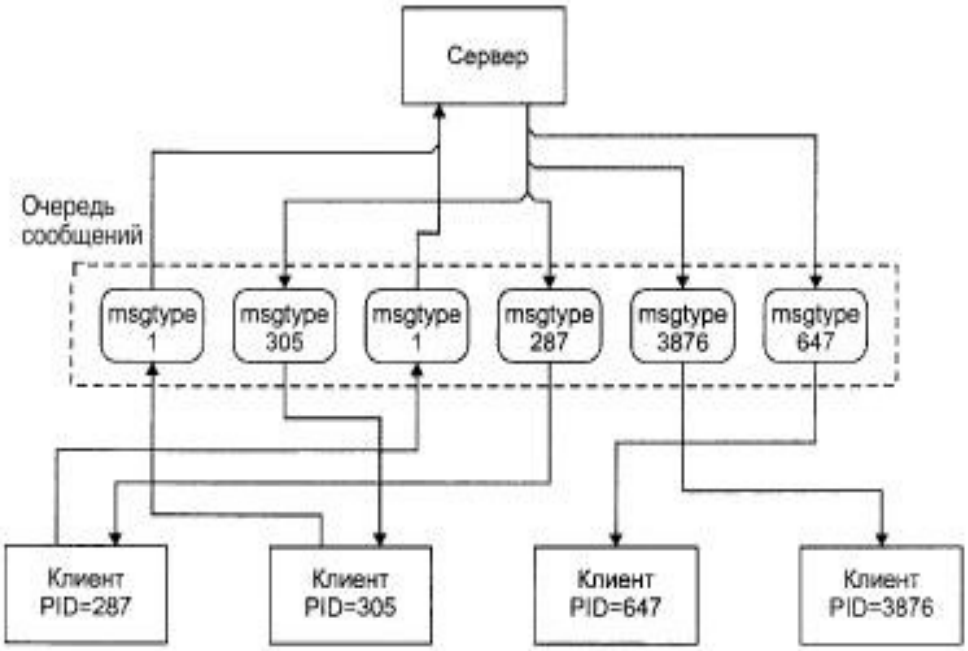
Очередь сообщений хранится в виде внутреннего однонаправленного связанного списка в адресном пространстве ядра.
Для каждой очереди ядро создает заголовок очереди *msgid_ds* , где содержится информация о правах доступа к очереди *msg_perm* , ее текущем состоянии (*msg_cbytes* - число байтов и *msg_qnum* - число сообщений в очереди), а также указатели на первое (*msg_first*) и последнее (*msg_last*) сообщения, хранящиеся в виде связанного списка.
Каждый элемент этого списка является отдельным сообщением.



После создания очереди сообщений процессы получают возможность коммуникации посредством следующих системных вызовов:

msgsnd() - поместить в очередь сообщение,
msgrcv() - получить сообщение,
msgctl() - управление сообщениями.

Очереди сообщений обладают полезным свойством - в одной очереди можно мультиплексировать сообщения от различных процессов. Для демультиплексирования используется атрибут *msgtype*, на основании которого любой процесс может фильтровать сообщения из очереди с помощью функции *msgrcv().*



В файлах *mq_local, mq_server, mq_client* содержится пример организации клиент-серверной коммуникации на основе очереди сообщений.
Клиентский процесс воспринимает ввод пользователя с клавиатуры и пересылает эту информацию вместе со своим идентификатором процесса в очередь сообщений.
Сервер читает сообщения из очереди, выполняет преобразование данных, введенных на клиенте с клавиатуры,
и отправляет модифицированные данные сообщением обратно клиенту.
Переправка серверу идентификатора клиентского процесса и задание сообщениям определенного типа для определенного клиента позволяют серверу вести обмен сразу с несколькими клиентами одновременно.

```
/* Файл   mq_local.h */
/*
 * Общий заголовочный файл для примера программы
 *        Message Queue Client-Serve
 */
 #include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <ctype.h>
#include <unistd.h>
#include <errno.h>
```

```
#define SEED  'g'       /* Заготовка для  ftok  */
#define SERVER 1L    /* Сообщение для сервера */

typedef struct {
  long        msg_to;
  long        msg_fm;
  char        buffer[BUFSIZ];
}MESSAGE;

/* Программа  mq_client.cpp */
/*
 *  Клиент  -  отправляет сообщения серверу
 */
#include "mq_local.h"

main(void){
   key_t        key;              /* Ключевое значение для ftok */
   pid_t        cli_pid;          /* Идентификатор процесса Process ID */
   int          mid, n;           /* Идентификатор очереди сообщений Message queue ID  */
   MESSAGE      msg;              /* Структура сообщения */
   static char   m_key[10];     /* Для символьной версии Message queue ID */
 cli_pid  =  getpid();
   if ((key = ftok(".", SEED)) == -1) {          /* Генерация ключа */
     perror("Client: key generation");
     exit(1);
   }
                 /* Создание очереди сообщений и получение доступа  */
   if ((mid = msgget(key, 0)) == -1)  {
       mid = msgget(key, IPC_CREAT | 0660);
        switch    (fork()) {
        case -1:
            perror("Client: fork");
            exit(3);
        case  0:
            sprintf(m_key, "%d", mid);  /* Перевод в строку символов */
            execlp("servermq.out", "servermq.out", m_key, "&", 0);
            perror("Client: exec");
            exit(4);
        }
   }
   while (1)  {
     msg.msg_to = SERVER;             /* Тип сообщения */
     msg.msg_fm = cli_pid;            /* Связывание с PID клиента */
     write (fileno(stdout), "cmd>", 6);/* Подсказка */
     memset(msg.buffer, 0x0, BUFSIZ);         /* Очистка буфера */
     n = read(fileno(stdin), msg.buffer, BUFSIZ);
     if (n == 0)                      /*  EOF ? */
       break;
     if (msgsnd(mid, &msg, sizeof(msg), 0) == -1)   {
       perror("Client: msgsend");
       exit(5);
         }
```

```
     if ((n = msgrcv(mid, &msg, sizeof(msg), cli_pid, 0))  !=  -1)
        write(fileno(stdout), msg.buffer, strlen(msg.buffer));
           }
    msgsnd(mid, &msg, 0, 0);
    exit(0);
}

/* Программа mq_server.cpp */
/*
 *  Сервер -  получает сообщения от клиентов
 */
#include "mq_local.h"

main(int argc, char *argv[])  {

   int         mid, n;
   MESSAGE     msg;
void        process_msg(char *, int);

   if (argc != 3)  {
      fprintf(stderr, "Usage: %s msq_id &\n", argv[0]);
      exit(1);
   }
   mid = atoi(argv[1]);                    /* Идентификатор очереди сообщений */
                                  /*        как параметр командной строки */
   while (1)  {
      if ((n = msgrcv(mid, &msg, sizeof(msg), SERVER, 0)) == -1)  {
         perror("Server: msgrcv");
         exit(2);
      } else if (n == 0)                   /* Клиент отработал */
        break;
       else {                               /* Обработка сообщений */
          process_msg(msg.buffer, strlen(msg.buffer));
          msg.msg_to = msg.msg_fm;     /* Свопинг сообщений:  to <-> from */
          msg.msg_fm = SERVER;
          if (msgsnd(mid, &msg, sizeof(msg), 0) == -1)  {
             perror("Server: msgsnd");
             exit(3);
                }
            }
    }                                       /* Удаление очереди сообщений */
    msgctl(mid, IPC_RMID, (struct msqid_ds *) 0);
    exit(0);
}
          /* Перевод строчных символов сообщения в прописные */
 void
 process_msg(char *b, int len) {
     int      i;
     for (i = 0; i <len; ++i)
       if (isalpha(*(b + i)))
         *(b + i) = toupper(*(b + i));
}
```